IMPROVED DIAGNOSTIC MONITOR FOR USE WITH AN OPERATING SYSTEM AND METHODS THEREFOR

By Inventor

John W. Curry

Related Applications

The present application is a continuation-in-part of and claims priority under 35 USC 120 to a co-pending, commonly-assigned US Application No. 10/376,436 entitled "IMPROVED DIAGNOSTIC EXERCISER AND METHODS THEREFOR" filed February 27, 2003 by inventor John W. Curry and incorporated by reference herein.

BACKGROUND OF THE INVENTION

[0001] As computers and operating systems become more complex, failure and performance analysis become more difficult. For reliability reasons, it is highly desirable to exhaustively test computer systems prior to introducing them into service. Generally speaking, it is possible to create codes to test specific portions of the computer system hardware. However, such narrowly focused tests do not stress the entire computer system as a whole, and do not reflect the way in which the computer system tends to be used in the field.

[0002] Another approach involves employing diagnostic tools, such as diagnostic exercisers, that execute under operating systems to detect errors. Nowadays, diagnostic exercisers are typically designed to execute under a production operating system ("OS"), e.g., Linux, HPUX™ (Hewlett-Packard Company, Palo Alto, CA) or Windows (Microsoft Corporation, Redmond, WA), and the like. By employing specifically designed applications that use special drivers and/or operating system calls, diagnostic exercisers executing under a production OS can stress the computer system in specific ways and detect certain errors when they occur. Because these diagnostic exercisers employ the OS, their applications can stress the hardware more fully and in ways that are more similar to the way the computer system would be used

in the field, e.g., involving features such as multi-threading, rapid disk I/O operations, and the like.

[0003] In general, the amount of stress experienced by the computer system can vary depending on which diagnostic exerciser applications are being executed. When a given diagnostic exerciser executes under a production operating system, such diagnostic exerciser is however restricted in capabilities by the operating system. For example, due to competitive reasons, production operating systems tend to be optimized for performance in the field and not for extensive diagnostic capabilities. Thus, a diagnostic exerciser executing under such a production OS tends to be restricted in its diagnostic capability. Analogously, other features useful for data gathering, error detection, and error analysis may not be available in a production OS since the provision of such features would unduly hamper OS performance and/or compromise the security of the computer system in the field.

[0004] Furthermore, most diagnostic exercisers executing under a production OS tend to run as privileged processes and/or drivers. This fact tends to limit their usefulness in a kernel crash situation. For example, if the kernel crashes during kernel booting, the common diagnostic monitor of the prior art diagnostic exerciser would not have been started, thereby being of little value for the detection and analysis of kernel boot failures.

[0005] Additionally, when the kernel crashes, most common diagnostic techniques involve obtaining a kernel dump and analyzing the kernel dump data. However, such diagnostic techniques are inherently limited by the information saved in the kernel dump file. If the programmer who originally designed the kernel dump program decided not to save certain information, such information will not be available in the kernel dump file for analysis. Furthermore, the content and format of the information available in the kernel dump file may vary depending on the discretion of the programmer who originally designed the kernel dump. Accordingly, the kernel dump information may be difficult to decipher, and it may be necessary to find the programmer who originally designed the kernel to obtain assistance in analyzing the kernel dump file. If a long period of time has elapsed between the time the OS is produced and the time a particular error occurs that results in the kernel dump, the

programmer may no longer be available to assist, and the use of kernel experts maybe required to sift through the kernel dump data.

SUMMARY OF THE INVENTION

[0006] The invention relates, in one embodiment, to a computer-implemented method for monitoring a computer system when the computer system executes a user application using a production operating system (OS). The OS has a OS kernel with a kernel trap arrangement. The method includes providing a diagnostic monitor, the diagnostic monitor being configured to be capable of executing even if the OS kernel fails to execute, the diagnostic monitor having a monitor trap arrangement. If a trap is encountered during execution of the user application, the method includes ascertaining using the diagnostic monitor whether the trap is to be handled by the OS kernel or the diagnostic monitor. If the trap is to be handled by the OS kernel, the method includes passing the trap to the OS kernel for handling.

[0007] In another embodiment, the invention relates to an arrangement, in a computer system,for diagnosing a computer system while executing a user application program. The user application program is executed under a production operating system (OS) kernel. The arrangement includes a diagnostic monitor configured to execute cooperatively with the OS kernel, the OS kernel having a kernel trap arrangement for handling at least one of a trap-type message and an interrupt-type message generated during the execution of the application program, the diagnostic monitor being capable of continuing to execute even if the OS kernel fails to execute. The diagnostic monitor includes a monitor trap arrangement for handling at least one of the trap-type message and the interrupt-type message. The diagnostic monitor is configured to receive traps generated during the execution of the application program and decide, for a given trap received, whether the OS kernel would handle the trap received or whether the diagnostic monitor would handle the trap received.

[0008] In yet another embodiment, the invention relates to an article of manufacture comprising a program storage medium having computer readable code embodied therein, the computer readable code being configured to handle errors in a computer

system. The article of manufacture includes computer readable code implementing a diagnostic monitor. The diagnostic monitor is configured to execute cooperatively with a production operating system (OS) kernel in the computer system and configured to be able to execute even if the OS kernel fails to execute. The diagnostic monitor includes a monitor trap arrangement configured to receive traps generated in the computer system. The OS kernel has a kernel trap arrangement configured to handle traps passed from the diagnostic monitor. The article of manufacture includes computer readable code for loading the diagnostic monitor at system startup prior to loading the OS kernel.

[0009] These and other features of the present invention will be described in more detail below in the detailed description of the invention and in conjunction with the following figures.

BRIEF DESCRIPTION OF THE DRAWINGS

[0010] The present invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements and in which:

[0011] Fig. 1 illustrates, in accordance with one embodiment of the present invention, a diagnostic environment that includes computer hardware under diagnostic, and a diagnostic exerciser.

[0012] Fig. 2 illustrates the diagnostic monitor in greater detail in accordance with one embodiment of the present invention.

[0013] Figs. 3A and 3B illustrate, in accordance with one embodiment of the present invention, the loading sequence upon initialization.

[0014] Figs. 4A and 4B illustrate, in accordance with one embodiment of the present invention, the logic flow for diagnosing the performance of a computer system by the diagnostic monitor.

[0015] Fig. 5 illustrates, in accordance with one embodiment of the present invention, the steps for handling a kernel panic.

[0016] Fig. 6 illustrates, in accordance with one embodiment of the present invention, the steps for handling a monitor call by a diagnostic application.

[0017] Fig. 7 illustrates, in accordance with one embodiment of the present invention, the steps for handling an error call by a diagnostic application.

[0018] Fig. 8 depicts, in accordance with one embodiment of the present invention, a diagnostic arrangement that employs the inventive diagnostic monitor in conjunction with a production OS.

[0019] Fig. 9 illustrates the diagnostic monitor of Fig. 8 in greater detail in accordance with one embodiment of the present invention.

[0020] Figs. 10A and 10B illustrate, in accordance with one embodiment of the present invention, the error handling steps by the diagnostic monitor.

[0021] Fig. 11 illustrates, in accordance with one embodiment of the present invention, the steps for handling a panic-related monitor call.

[0022] Fig. A1 shows, in accordance with one exemplary implementation of the present invention, the basic modules that make up the RAGE exerciser platform.

[0023] Fig. A2 shows, in accordance with one exemplary implementation of the present invention, the basic functionality of the loader.

[0024] Fig. A3 shows, in accordance with one exemplary implementation of the present invention, the major components of the Monitor.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0025] The present invention will now be described in detail with reference to a few preferred embodiments thereof as illustrated in the accompanying drawings. In the following description, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art, that the present invention may be practiced without some or all of these specific details. In other instances, well known process steps and/or structures have not been described in detail in order to not unnecessarily obscure the present invention.

[0026] In accordance with one embodiment of the present invention, there are provided diagnostic arrangements and techniques that combine the advantages offered by the pure off-line diagnostic approach, e.g., extensive access to and control of the computer hardware and ease of error detection and analysis, with the advantages offered by the on-line OS-based diagnostic approach, e.g., the ability to stress the hardware more fully and in ways that are more representative of the stress experienced by the computer hardware in the field.

[0027] As the terms are employed herein, a diagnostic tool is said to be off-line if it does not employ a production OS in its diagnostic tasks. Off-line diagnostic tools can have extensive access to and/or control of the individual hardware subsystems but their diagnostic approach tends to be based on localized stimulus-responses. Stimulus-response tests based on off-line diagnostic tools have been known to be unsuitable for efficiently detecting certain transient errors.

[0028] Further, the off-line diagnostic tools do not stress the computer system hardware in a comprehensive way nor do they stress the computer system hardware in ways that are representative of the stress experienced by the computer hardware in the field. Accordingly, it has been necessary to employ a large number of stimuli (and a large number of permutations thereof) in order ensure that tests performed cover most of the scenarios that the subsystems under test would likely experience in the field. On the positive side, since off-line diagnostic tools tend to test individual subsystems

of the computer system hardware based on stimulus-response tests and without having the OS acting as the intermediary, extensive access to and/or control of the hardware functionality is possible and error isolation and analysis tend to be less complex.

[0029] In contrast, a diagnostic tool is said to be on-line if it employs the production OS in its diagnostic task. When the diagnostic applications execute under the OS, control of the hardware is achieved in a less direct manner and access to hardware is much more restricted. However, the on-line diagnostic applications can more fully stress the computer system as a whole, employing OS-related features such as multi-threading, rapid I/O accesses, and the like. Because the on-line diagnostic applications execute under the OS, the computer system tends to be stressed in ways that are more similar to those experienced by the computer hardware in the field. Furthermore, the time spent in testing can be shorter with an on-line diagnostic tool since the tests can be more directed toward the likely and/or realistic usage scenarios instead of, as in the case with the off-line diagnostic exerciser, randomly testing various permutations, some of which may be unrealistic and/or may never be experienced by the computer system in the field.

[0030] The additional capabilities of on-line diagnostic tools, however, come at the cost of inflexibility and complexity. Since existing OS-based diagnostic tools tend to execute under a production OS (i.e., the OS employed in the field), security and performance considerations severely limit the diagnostic and/or analytical capabilities of the existing OS-based diagnostic tools. Depending on the design of the OS and/or the OS kernel, certain types of information may not be captured and/or be accessible for error detection and/or error analysis. Access to and/or control of the hardware may be severely restricted. Even if the information is captured, information such as kernel dump may be difficult to analyze to ascertain the source of error. OS-based diagnostic tools also tend to be difficult to isolate, in a concise way, the source of the error.

[0031] As mentioned earlier, embodiments of the inventive diagnostic exerciser combine the advantages offered by a pure off-line diagnostic approach with the advantages offered by the on-line OS-based diagnostic approach. To fully exercise the hardware system and achieve more directed testing of the computer system using tests that are more representative of the way the computer system is employed in the

field, the diagnostic exerciser employs the OS. However, the OS employed is a non-production OS based on a diagnostic kernel. The use of a non-production OS and more particularly a non-production kernel modified to enhance diagnostic capabilities advantageously allows the invention to avoid the limitations of the production OS, which limitations are imposed based on, for example, security and/or performance reasons. Since diagnostic testing is performed in a controlled, non-production environment, security and performance in the field considerations are less important. Furthermore, a diagnostic monitor is provided to monitor and to work cooperatively with the non-production kernel. As the term is employed herein, the diagnostic monitor is said to execute cooperatively with the non-production kernel since it does not operate under the non-production kernel. There are furnished, in accordance with certain embodiments of the invention, rule-based facilities to provide failure analysis in an easy to understand manner. Inventive techniques are employed to facilitate failure analysis even if the kernel crashes during booting or during execution of the diagnostic application programs.

[0032] In a preferred embodiment, a diagnostic monitor having detailed information about the kernel (i.e. data structures, kernel data locations, etc) is loaded into a memory area that is not used by the OS. The diagnostic monitor is loaded prior to kernel loading and booting and has control of the trap vector table. During the kernel boot, or while the kernel (i.e. OS) is running, any traps or interrupts that occur would be examined by the diagnostic monitor to determine how the traps or interrupts should be handled.

[0033] To elaborate, the trap/interrupt mechanism of modern computer architectures allows privileged code to run when certain conditions occur. Depending on the particular architecture, these conditions include certain hardware events and some software generated events. The trap/interrupt-based mechanism is exploited by the inventive diagnostic monitor to facilitate error detection as well as data collection and analysis pertaining to certain errors.

[0034] In general, all non-error traps would be passed by the diagnostic monitor to the normal kernel vector table. Otherwise, the diagnostic monitor would examine the trap information, along with any information it can gather from the kernel data structures

in memory, and use this information for error isolation. Kernel panics may cause forced traps to the diagnostic monitor. For many architectures, process, time-slice events may cause non-error traps. These can be used by the diagnostic monitor to "monitor" hardware and kernel parameters at run-time. For example, if a CPU has a time-slice trap, the diagnostic monitor can determine whether or not that CPU was idle by examining the process data structure associated with the process being run on that CPU. If the CPU is idle, the diagnostic monitor can use it to check for errors. Once the checking is done, the CPU would be returned to its idle loop. In this way, the diagnostic monitor can lower its performance impact on the kernel.

[0035] In most common operating systems, semaphores are not timed. That is to say, if a kernel thread attempts to grab some semaphore, it will do so forever until it obtains the semaphore. However, if the kernel semaphore routines are altered so that they time out after a period of time, and force a trap, the diagnostic monitor can be used for analysis.

[0036] In accordance with one embodiment of the present invention, a loader program (the loader) is running initially. The loader then finds the diagnostic monitor from the boot device (such as a disk) and launches the diagnostic monitor. The diagnostic monitor initializes, and tries to determine what model of computer it is running on. In general, a given diagnostic monitor is compiled for particular machine architecture. The diagnostic monitor then requests that the loader to load the appropriate library module for the model of computer the diagnostic monitor is running on. The library has model-specific information about the computer being monitored, such as, the detection and isolation routines of various hardware errors. Furthermore, the library has detailed information about the operating system to be run.

[0037] The diagnostic monitor generally needs to know various OS details such as the layout of process data structures, where the OS will be loaded, etc. The symbol table of the kernel is also loaded and used by the diagnostic monitor so that the diagnostic monitor knows the locations of the fixed kernel data structures. The locations of most run-time kernel structures can be deduced by knowing the locations of certain fixed structures. For any exceptions, the kernel may pass the locations to the diagnostic monitor during OS initialization, or at the structure creation time. Additionally, the

diagnostic monitor installs its trap vector table as the one to be used. It then returns control to the loader.

[0038] The details and advantages of the invention may be better understood with reference to the drawings and discussions that follow. Fig. 1 illustrates, in accordance with one embodiment of the present invention, a diagnostic environment that includes computer hardware 102 under diagnostic, and a diagnostic exerciser 104.

[0039] Diagnostic exerciser 104 includes a loader module 105, which is responsible for loading major components of diagnostic exerciser 104 into memory upon system initiation. These major components are discussed hereinbelow.

[0040] Diagnostic exerciser 104 includes a diagnostic kernel 106, representing the kernel of the non-production OS (e.g., a non-production Windows™ or Linux OS) employed for diagnostic purposes. There are provided with diagnostic kernel 106 a plurality of drivers 107a, 107b, 107c, representing some of the drivers provided to allow the diagnostic exerciser 104 to obtain information pertaining to I/O devices. Thus, when an I/O device fails, instrumentation may be performed on the associated driver to gather information from the card/device electronics to allow diagnostic exerciser 104 to analyze the error.

[0041] Generally speaking, it is preferable to keep the changes between diagnostic kernel 106 and the kernel of the production OS minimal in order to allow computer hardware 102 to be stressed in ways that are representative of the stress experienced by computer hardware 102 in the field. However, certain modifications will be required to allow the diagnostic monitor (discussed below) to perform its functions of monitoring kernel 106, error detection, and error analysis. In one embodiment, kernel 106 is modified to allow the initialization of the diagnostic monitor and the drivers, to facilitate communication with the diagnostic monitor during execution of the diagnostic applications, and to provide for additional diagnostic capabilities (such as processor affinity, physical memory allocation, and the like). Other possible changes to the production kernel are discussed further hereinbelow.

[0042] Diagnostic exerciser 104 also includes a diagnostic monitor 108, representing the monitoring software that is involved in detecting and analyzing errors. Diagnostic

monitor 108 is designed to communicate with and to monitor kernel 106. In one embodiment, monitor 108 and kernel 106 are furnished as separate binary files, thereby rendering monitor 108 independent of the OS (i.e., monitor 108 runs with rather than runs under kernel 106). Monitor 108 is preferably loaded into a memory location that is unused by kernel 106 to avoid any unintended alteration of monitor 108 by kernel 106. When an error occurs with kernel 106, monitor 108 will attempt to analyze the error and isolate the problem to a FRU (field replaceable unit) or a list of potential offending FRUs. In this sense, the inventive diagnostic exerciser provides the error isolation granularity typically associated with off-line diagnostic tools.

[0043] As shown in Fig. 1, diagnostic monitor 108 includes three main modules: monitor logic 110, monitor library 112, and monitor trap table 114. Monitor logic 110 is architecture-dependent (e.g., PARISC™, Itanium™, etc.). Depending on the specific architecture and the type of box of computer hardware 102 (the information pertaining to which may be obtained via link 116), the appropriate monitor library 112 would be loaded. Monitor library 112 is thus both architecture-specific and box-specific in one embodiment.

[0044] Monitor trap table 114 represents the trap table that is in control during execution of the diagnostic applications. As will be discussed later herein, one of the major functions of diagnostic monitor 108 is to control the trap handler. Any trap encountered by kernel 106 would be examined by monitor 108, which will decide whether it will handle the trap itself or will pass the trap on to the kernel for handling. The kernel also has a trap table, but the kernel's trap table is granted control only if diagnostic monitor 108 decides that it wants the kernel to handle a particular trap. In such case, control will pass to the appropriate offset in the trap table of the kernel to handle the trap. Thus, as far as the kernel is concerned, its own trap table gets the traps that the kernel needs to handle in a transparent manner. Other traps are retained by and handled by monitor trap table 114.

[0045] Applications 130, 132, 134, and 136 represent some diagnostic applications that may be executed under kernel 106 to stress specific portions of computer hardware 102. Thus CPU application 130 is designed to emphasize testing of the CPU; memory application 132 is designed to emphasize testing of the memory; and

I/O application 134 is designed to emphasize testing of the I/O subsystems. Other applications may be provided to test other subsystems and/or aspects of computer hardware 102 and/or the OS and are represented generically by application 136 in Fig. 1.

[0046] Fig. 2 illustrates the diagnostic monitor 108 in greater detail in accordance with one embodiment of the present invention. As can be seen, monitor logic 110 includes, in addition to core monitor codes 214, architecture routines 210 and rules engines 212. Although architecture routines 210 and rules engines 212 are shown separate from monitor logic 110 in Fig. 2 to facilitate discussion, they are in fact part of monitor logic 110. Architecture routines 210 represent routines that may perform standard tasks based on a given architectural level. For instance, the contents of all general registers may need to be saved. This would be done by an architectural routine. The size and number of general registers depends on the particular architecture under consideration (e.g. PARISC 2.0 has 64 bit width registers, while PARISC 1.0 has 32 bit width registers). In one embodiment, although the core monitor is dependent on the architecture, different monitor libraries are provided to accommodate different boxes, thereby enabling the core monitor to be employed for all boxes pertaining to a given architecture.

[0047] Monitor trap table 114 is designed to be the functional trap handler during execution. Thus, any trap/interrupt by kernel 106 (see Fig. 1), expected or not, will force it to enter diagnostic monitor 108. This feature allows diagnostic monitor 108 to be made aware of any error traps right away, at which point diagnostic monitor 108 may stop all other processors, and collect any state information. For non-error traps, for example, diagnostic monitor 108 may collect certain statistics (if desired) and then continue on with the basic function of the trap, eventually returning to the kernel.

[0048] Diagnostic monitor 108's trap handler is organized in such a way that kernel 106 can specify (for non-error traps) what routine diagnostic monitor 108 should run. This specification should occur during kernel initialization, once the kernel knows there is a diagnostic monitor 108. Otherwise, the kernel may use its default trap handler.

[0049] Certain error traps may need to be handled in a special way. For example, for severe hardware errors, significant software errors, and hardware resets, the hardware may automatically branch to firmware routines. In the context of PARISC™ machines, for example, HPMCs (High Priority Machine Checks), LPMCs (Low Priority Machine Checks), and TOCs (Transfer of Controls) may cause the hardware to automatically branch to firmware routines. The firmware may then save the register state of the processor and clear any error conditions. If the appropriate trap handler routine in diagnostic monitor 108 is setup properly, the firmware routine may branch to diagnostic monitor 108's handler. At this point, diagnostic monitor 108 can then begin analysis.

[0050] Rules engine 212 of monitor logic 110 includes a set of routines to interrogate monitor library 112 in order to determine how to find the FRU for some error condition. Rules engine 212 is employed to interpret rules 220 (described below), and may also include functionality such as which rules will be employed for error analysis and when error analysis is completed. There may be a library for each type of "box" for a given architecture. The library may contain (among other things) rule functions that may analyze collected data in order to figure out what went wrong. The interface to the rules may be the same across different architectures.

[0051] Monitor library 112 includes, in addition to core library codes 224, rules 220 and hardware-specific routines 222. Rules 220 represent rules specific to the architecture under consideration and are employed for error analysis. Rules 220 has the inferences for various error conditions and can resolve the error to the failing FRU. Hardware-specific routines 222 include routines specific to a particular box to facilitate, for example, the gathering of any necessary information from the specific box.

[0052] Upon initialization of the computer system, loader module 105 loads the components of diagnostic exerciser 104 in a particular sequence. Fig. 3 illustrates, in accordance with one embodiment of the present invention, the loading sequence upon initialization. In block 302, the loader ascertains whether a diagnostic monitor (such as a binary image of diagnostic monitor 108) is present on the boot device, e.g., a hard disk, a CDROM, or even a particular storage location in a network. If the diagnostic

monitor is not present on the boot device, the loader ascertains in block 304 whether the kernel (such as kernel 106) is present on the boot device. If there is no kernel on the boot device, the loading process indicates failure in block 306.

[0053] On the other hand, if there is a kernel on the boot device (as ascertained in block 304), the kernel is loaded, and control is passed to the kernel in block 308. In this case, there is no monitor (as determined earlier in block 302) and thus in block 310, the kernel would ascertain that there is no diagnostic monitor present. Note that this check for the presence of a diagnostic monitor is part of the modifications made to the production OS. With no diagnostic monitor present, the kernel executes normally, with the kernel's trap table in control.

[0054] If there is a diagnostic monitor on the boot device (as ascertained in block 302), the loader then loads, in block 330, monitor logic 110 of diagnostic monitor 108 into memory. Thereafter, monitor logic 110 is initialized in block 332 to, for example, ascertain the box information for the hardware being tested using the associated architectural routines (such as architectural routines 210 of Fig. 2). The box information is then passed to the loader to enable the loader to ascertain, in block 334, whether the requisite monitor library is present on the boot device or another device. If the required monitor library is not present, the process of loading the monitor fails. In that case, the method returns to block 304 to begin loading the kernel, i.e., treating the situation from that point on as if the monitor was not present on the boot device.

[0055] On the other hand, if the required monitor library is found (as ascertained in block 334), the required monitor library is loaded in block 336. After the required monitor library is loaded and synchronized for communication with monitor logic 110 (using, for example, dynamic linking technologies), control is passed back to the loader, which then ascertains in block 338 whether the kernel is on the boot device. Note that at this point, the diagnostic monitor is already loaded and initialized. Accordingly, if the kernel is not found (block 338) or if the kernel fails to boot or if there is another problem with the kernel, diagnostic monitor can detect, analyze, and report the error. This is unlike the situation with existing diagnostic tools which run under production OS's. In the case where an existing diagnostic tool runs under a

production OS, if the OS fails to get loaded, the diagnostic tool would not have been initialized and would therefore have been of no value in detecting, analyzing, and/or reporting the error.

[0056] If the kernel is not found in block 338, the loading process reports failure in block 306 since there can be no application execution under the OS if the kernel cannot be found. On the other hand, if the kernel is found, the kernel symbol table is loaded in block 340. Generally speaking, the symbol table provides addresses of important elements (such as data structures and subroutines) of the kernel. The diagnostic monitor relies on the kernel symbol table, which is embedded in a non-stripped version of the kernel binaries. Thus, in one embodiment, the loader obtains the symbol table from the kernel ELF binary and loads the symbol table into memory.

[0057] Once the symbol table is loaded in block 340, the kernel is loaded in block 308. The kernel is preferably loaded into a fixed address in memory. In block 310, the kernel ascertains whether the monitor is present. In one embodiment, the information regarding the presence of the monitor is passed from the loader to the kernel. If the diagnostic monitor is not deemed present for whatever reason (as ascertained in block 310) the kernel executes normally in block 312 with its trap table in control. On the other hand, the diagnostic monitor is present (as ascertained in block 310), the kernel may optionally call upon the diagnostic monitor (in block 314) to, for example, ascertain which traps are handled by the diagnostic monitor's trap table and which traps are handled by the kernel's trap table. Of course the decision pertaining to which traps are handled the kernel and which traps are handled by the diagnostic monitor may be worked out in advance and hard-coded. In such a case, optional step 314 may be omitted.

[0058] Once the kernel completes initialization with the diagnostic monitor, the kernel runs with the diagnostic monitor, and the monitor trap table in control (block 316).

[0059] Fig. 4 illustrates, in accordance with one embodiment of the present invention, the error handling steps by the diagnostic monitor. Because a trap could be a result of normal operation or it could represent a problem, the diagnostic monitor needs to

examine traps and make certain decisions regarding how the traps are handled. In block 402, the diagnostic monitor ascertains whether the trap is to be handled by the kernel or by the diagnostic monitor itself. In one embodiment, the decision is made using a bit masking technique for looking at the bit in a particular position of the variable. Other techniques may also be employed, including the use of arrays.

[0060] If the trap is a result of normal operation, such as a page miss, such trap may be handled by the OS's kernel. In that case, the method proceeds to block 404 wherein it branches to the appropriate OS trap table offset using the symbol table loaded earlier (e.g., in block 340 of Fig. 3) and/or the information obtained during diagnostic initialization call by the kernel (e.g., in block 314).

[0061] For example, in the PARISC™ implementation, branching can be performed by, for example, using a relative branch instruction (e.g., branch to five instructions away) or by giving the actual address to branch to in the branch instruction. Alternatively, branching may be accomplished by reference to a general register, which contains the address.

[0062] However, for Itanium-based systems, if a branch instruction employs a branch register, there is a potential for inadvertently overwriting the content of a branch register that is in use by the kernel since the kernel also employs branch registers. It is possible to force the kernel to temporarily store and restore the values of these registers but that may involve certain additional modifications to the production kernel.

[0063] To keep the number of changes to the kernel low, there is provided, in accordance with one embodiment of the present invention, a long branch table within the diagnostic monitor. With reference to the IPF™ implementation, for example, the branch table is initially setup so that all branches jump to themselves. During initialization, the monitor can dynamically calculate the address of the long branch instructions and can alter all the branches of the branch table to branch to the appropriate offset in the kernel trap table. This is possible because the kernel trap table resides at a fixed address.

[0064] To clarify, a long branch is a type of relative branch that allows one to branch to a long offset, typically longer than normally performed using a normal relative branch. Using long branches allows the monitor to be positioned at any desired location in memory, and still be able to get to the kernel trap handler. Otherwise, the monitor would have to reside near the kernel's trap handler in memory. Furthermore, the monitor long branch table renders it possible to avoid using the branch registers, which are utilized by the kernel. Thus, it is not necessary to keep track of the contents of the branch registers, which would require more kernel modification. For implementations that do not involve branch registers, for example PARISC™, general registers may be employed instead.

[0065] Returning to Fig. 4, if the trap is to be handled by the diagnostic monitor (as ascertained in block 402), a decision is made in block 406 whereby the diagnostic monitor ascertains whether the trap is an explicit monitor call by the kernel. By way of example, the immediate values associated with break trap instructions, which are employed to monitor calls, may be specified to allow the diagnostic monitor to ascertain whether a trap is an explicit monitor call by the kernel. If the trap is an explicit monitor call, it is handled in block 440, which is discussed in connection with Fig. 6 hereinbelow.

[0066] If the trap is not an explicit monitor call (as determined in block 406), the hardware-triggered error is assessed in block 408 of Fig. 4B to determine whether the error is severe. As an example, the failure of kernel boot is a severe error. In IPF™, for example, the occurrence of a Machine Check Abort (MCA) condition is typically regarded as a severe error. In PARISC™, for example, the occurrence of a High Priority Machine Check (HPMC) condition is typically regarded as a severe error. Generally speaking, each architecture has its own way of representing a severe error. However, details on the error can vary between different box types. For example, there may be one bit in a given hardware register in the processor which will be set if the error is severe. The fact that the diagnostic monitor has direct access to the computer hardware facilitates the assessment in block 408. Generally speaking, the error details may be ascertained using the rules.

[0067] If the error is deemed severe (as ascertained in block 408), the diagnostic monitor then stops the kernel from running by sending an interrupt signal to all other processors (assuming there are multiple processors). It is possible, for example, for one processor to experience a severe error (e.g., a HPMC in the case of PARISC™) but other processors of the computer system may not have experienced the same severe error. Freezing the processors by an external interrupt signal allows the state of the processors to be collected in block 412 to facilitate analysis in block 414. Of course if there is only one processor in the system, there is no need to send the interrupt signal to the non-existent other processors, in which case block 410 is not necessary.

[0068] In block 414, the error analysis may be accomplished using, for example, the hardware-specific routines (222 in Fig. 2) of the monitor library. The hardware-specific routines may examine various registers of the processors, the I/O registers, certain data structures in the kernel, and the like, to ascertain the error and/or the cause thereof. The analysis result is displayed in block 416. In one embodiment, the result may be displayed in a monitor shell that is separate from the OS's shell. The monitor shell permits certain diagnostic-related inputs and outputs and can continue to run even if the OS crashes. For example, one can employ the monitor shell to specify certain kernel data structures and routines and causing those to be displayed for examination. This is represented in block 418 of Fig. 4.

[0069] On the other hand, if the error is not deemed severe (by block 408), the state information pertaining to the error can be collected in block 430. In block 432, analysis is performed using the rules and hardware-specific routines of the library to narrow the error down to one or more FRUs (Field Replaceable Unit). For example, if a single bit error is encountered, block 432 may attempt to narrow the error down to a single DIMM and/or a single memory slot.

[0070] Note that the rules and rules engine of the monitor allow it to analyze the relevant (box specific, in some cases) registers and data structures of the kernel (if necessary) and to narrow the problem to, for example, the failing FRU. This is in contrast to the prior art situation employing a standard OS, wherein an experienced user or expert would have to manually analyze the data dumped by the kernel.

Furthermore, the monitor has access to all system and kernel data, thereby ensuring that the monitor can obtain any required information in order to accurately perform the error detection and error analysis tasks. In contrast, the data dump by the kernel may not, in some cases, contain all the required data for error analysis.

[0071] The information pertaining to the error is logged in block 434. The logged errors may be retrieved by an application program at a later point in time (e.g., for further analysis or display) using a monitor call. In block 436, since the error is non-severe, the method preferably continues with the next instruction and returns to the caller to allow execution to continue.

[0072] There are times when kernel panics occur without traps. For example, the kernel may have encountered some illegal situation where it runs out of resources (e.g., memory) or experiences a problem that it cannot resolve (e.g., an unresolvable page fault). In this case, the kernel may have made a panic call, which is converted into a monitor call by the non-production kernel to allow the diagnostic monitor to become involved. The kernel panic-related monitor call however contains information identifying the monitor call as one that results from a kernel panic. Once the panic-related monitor call is received, it is handled in blocks 502-510 of Fig. 5. Blocks 502-510 perform error handling in an analogous fashion to that performed by blocks 410-418 of Fig. 4.

[0073] Fig. 6 shows, in accordance with one embodiment of the present invention, block 440 of Fig. 4 in greater detail. In this case, the monitor call is neither occasioned by the hardware nor caused by a kernel panic. Rather, an application has made a monitor call to report an error or, for example, to inquire about errors. Thus, in block 602, the diagnostic monitor determines whether the monitor call is simply a normal monitor call or a monitor error call, i.e., the application has already determined that there is an error. An error monitor call may be made if, for example, the application has determined that the ECC (Error Correcting Code) circuitry is defective. If the monitor call is an error monitor call, the method proceeds to block 604 wherein the error monitor call is handled in a subsequent Fig. 7, which is discussed later herein.

[0074] On the other hand, if the monitor call is not an error monitor call, the method proceeds to block 606 wherein the diagnostic monitor ascertains the nature of non-error monitor call received. In one embodiment, this is accomplished by consulting the call index in a table. An example of such a non-error monitor call made by the application may involve a call to determine the number of processors in the system, or a call to determine if any correctable errors had occurred.

[0075] Once the diagnostic monitor ascertains the nature of the non-error monitor call received, the action requested in the non-error monitor call is performed in block 608. In block 610, the information requested in the non-error monitor call, if any, is returned to the calling application.

[0076] Fig. 7 shows, in accordance with one embodiment of the present invention, block 604 of Fig. 6 in greater detail. In Fig. 7, an error monitor call has been received, which is different from the hardware generated trap handled by Fig. 4. In this case, the application has determined that there is an error with the hardware, and the diagnostic monitor proceeds to collect data and perform error analysis to validate the reported error monitor call. Thus, in block 702 the diagnostic monitor will log the received error monitor call, along with any data furnished by the application. In block 703, the error monitor call is analyzed using, for example, the rule-based analysis. In block 704, the diagnostic monitor will ascertain whether the error monitor call relates to a severe error. A HPMC (PARISC™) or a MCA (IPF™) represents an example of a severe error that causes explicit types of traps. An example of a severe, non-trapping type of error may be, for example, a semaphore timeout. In a semaphore timeout, some processor has grabbed the semaphore for longer than some certain threshold, which situation indicates an imminent hang. In this case, the semaphore may timeout, for example, and result in a call to the monitor.

[0077] If the error monitor call is deemed severe in block 704, the diagnostic monitor would treat the severe error monitor call in the same manner that it treats kernel panics. Thus, the method proceeds to block 708 wherein the severe error monitor call is handled in the manner discussed earlier in connection with Fig. 5.

[0078] On the other hand, if the error monitor call relates to a non-severe error, the method proceeds to block 706 to return to the calling routine (since the error monitor call has already been logged in block 702) to continue execution.

[0079] As can be appreciated from the foregoing, the inventive diagnostic exerciser combines the best of the off-line diagnostic exerciser approach with the best of the on-line diagnostic exerciser approach. By using a non-production kernel, limitations imposed by production OS are effectively removed, allowing the inventive diagnostic exerciser to have more direct access to the computer hardware and information that would have been otherwise unobtainable from a production kernel. This enhanced access to the computer hardware and kernel data structures and subroutines enhances the error detection and error analysis capabilities of the inventive diagnostic exerciser. However, the use of a non-production OS that has minimal difference from the production OS (with the difference being those made to accommodate the diagnostic monitor), the inventive diagnostic exerciser can stress the computer hardware more fully, accommodate more directed tests with appropriately designed applications (which can substantially reduce the required diagnostic testing time) as well as accommodate tests that are more reflective of the way the computer hardware may be used in the field.

[0080] Furthermore, the provision of a diagnostic monitor that runs in cooperation with, instead of under, the non-production OS enables the diagnostic monitor to be able to monitor the kernel failure and to capture information pertaining to kernel failures for analysis. Since the diagnostic monitor is loaded before the loading of the non-production kernel, diagnosing is possible even if the kernel fails during loading. If an error occurs, the provision of a rule-based analysis system facilitates the analysis and display of error and/or source thereof in an easy-to-understand manner.

Exemplary Implementation.

[0081] In the example discussed below, an exemplary implementation of a diagnostic exerciser based on the Linux operating system is discussed. It should be kept in mind that the exemplary implementation of Appendix A is but one implementation of the invention, and design choices, assumptions, and limitations suggested for this

particular implementation should not be construed in a blanket manner to be limitations of the invention, which are clearly defined by the claims herein.

[0082] In the exemplary implementation below, Rage is the name given for non-production kernel and Ragemon is the name given for the diagnostic monitor.

[0083] INTRODUCTION

[0084] FUNCTIONALITY/FEATURES

[0085] RAGE is offline exerciser based on LINUX. It is designed, to be instrumented in such a way as to allow it to be monitored by a separate executable. The applications that run under RAGE are designed to stress the hardware, as well as perform some diagnostic capabilities. RAGEMON is the executable designed to monitor RAGE. Both RAGE and the Monitor are loaded into memory by a loader ELILO. Which is a modified version of the standard LINUX loader.

[0086] Rage intends to implement the following features:

[0087] To extend the coverage of our traditional "point to point" tests.

[0088] To better capture transient errors by generating more stress patterns.

[0089] To have the ability to detect and recover from hardware errors (i.e. MCAs).

[0090] To have the ability to isolate hardware errors to a FRU or at least a small list of FRUs.

[0091] The target platform is Orca

[0092] The following list of features would also be desirable to have in the Offline exerciser:

[0093] Test suites that cover all the major hardware elements.

[0094] The ability to add "point to point" tests directed at a specific hardware target.

[0095] Platform portability.

[0096] The ability to specify what stress tests suites should run and on what targets.

[0097] Fast loading with a small initial footprint.

[0098] Monitor is still alive if kernel crashes.

[0099] DEPENDENCIES

[00100] In general RAGE has the following dependencies:

[00101] The most obvious dependency is of course the LINUX kernel itself.

[00102] EFI Runtime services

[00103] Firmware, PAL & SAL calls

[00104] Appropriate compilers

[00105] ASSUMPTIONS

[00106] It is assumed that the LINUX kernel will have support for Orca (can boot, and run, etc.)

[00107] DESIGN OVERVIEW

[00108] DESIGN CONTEXT

[00109] Initially, RAGE can be used in the offline environment after running the ODE based diagnostics. After RAGE is tuned, it can eventually replace many of the standard offline diagnostics, just leaving a core set of ODE based tests. This small core set of tests would primarily focus on the CPU for cache pattern tests and TLB functionality. Then, RAGE can be launched to stress test the rest of the system. Eventually, RAGE should be able to replace the standard online testing as well. The final desired scenario would be: 1) core ODE tests; 2) RAGE tests.

[00110] OVERVIEW OF OPERATION

[00111] The diagram of Fig. A1 shows the basic modules that make up the RAGE exerciser platform.

[00112] As the diagram illustrates, RAGE is made up of 3 components: the Kernel, the Monitor, and the RAGE applications. Note that the kernel and Monitor are at the same level. The RAGE Monitor does not run under the kernel, but rather with the kernel. The RAGE applications run under the kernel, as one would expect. And they actually perform the hardware stress. The applications are analogous to the ODE diagnostics under the ODE platform.

[00113] MAJOR MODULES

[00114] The following sections give a more detailed explanation of the Major RAGE modules.

[00115] LOADER

[00116] The basic functionality of the loader is shown in the diagram of Fig. A2.

[00117] The kernel loader will be responsible for loading all the Rage modules into memory. The first module it will load will be the Monitor. Once loaded, the Monitor will do some initialization (number 1 in Fig. A2), and then determine what hardware it is running on. Based on that information, the Monitor will inform the loader which Library to load.

[00118] The library is a "box version" specific module that contains specialized information, such as how to grab the contents of certain diagnose registers. The Monitor itself, is only architecturally specific. The Libraries should be viewed as a collection of special routines for the particular type of box we are running on. Once the proper library is loaded, it will initialize with the Monitor (number 2 in Fig. A2). Initialization would include such things as, letting the Monitor know where the library entry point is.

[00119] Once the Library is loaded, the Loader will load the RAGE kernel. The loader will pass the address of the Monitor's trap handler to the kernel. If there is no Monitor present on the. boot device, the loader will pass zero as the address of the RAGE Monitor's trap handler. In this way, the kernel will know whether to use its

own internal trap handler, or the Monitor's. If the Monitor's trap handler address is non-zero, the kernel will assume that a Monitor is present (see number 3 in Fig. A2).

[00120] MONITOR

[00121] RAGEMON's job is to determine what went wrong with the kernel when an error occurs, and to attempt to isolate the problem to a FRU (or at least an ordered list of FRUs) for replacement. To do this, RAGEMON will have information from the kernel. For example, RAGEMON will want to know what range of memory is actually occupied by the kernel, as opposed to memory being available to processes. When the kernel first comes up and detects that a Monitor is present, it will initialize with the Monitor to pass certain important information. RAGEMON (if available) will be present while the kernel is running, and even while the kernel is booting. The kernel will always be loaded at a fixed address (pre-determined).

[00122] As can be seen from the figure of Fig. A3, the Monitor has several major areas that it is composed of. Architecture routines are routines that will perform standard tasks based on a given architectural level. For instance, the contents of all general registers will need to be saved. This will be done by an architectural routine. The size and number of general registers depends on the particular architecture one is using (e.g. PARISC 2.0 has 64 bit width registers, while PARISC 1.0 has 32 bit width registers). Note that the Monitor itself is also architecturally dependent. Thus, the Monitor for a PARISC 2.0 machine, will be different than one for an IA-64 machine. However, the basic functionality of the routines within each Monitor would be the same.

[00123] The trap handler within the Monitor is designed to be the functional trap handler while the RAGE kernel is running. Thus, any trap/interrupt by the kernel, expected or not, will force it to enter the Monitor. This feature allows the Monitor to be made aware of any error traps right away. At which point, the Monitor can stop all other processors, and collect any state information. For non-error traps, the Monitor could collect certain statistics (if desired) and then, continue on, with the basic function of the trap, eventually returning to the kernel. The Monitor will know the location of the kernel's trap handler from the kernel symbol table, so that it can branch

to the appropriate offset in the kernel trap table. Otherwise, the kernel will use its' default trap handler. Certain error traps need to be handled in a special way. For MCAs, and CMCs, the hardware will automatically branch to firmware routines. The firmware will then save the register state of the processor and clear any error conditions. If the appropriate trap handler routine in the Monitor is setup properly, and if the firmware deems the error not severe enough to require a reboot, the firmware routine will branch to the Monitor's handler. At this point, the Monitor can then begin analysis.

[00124] The rules engine of the Monitor is basically a set of routines that allows it to interrogate the Library in order to determine how to find the FRU for some error condition. There will be a library for each type of "box" for a given architecture. The library will contain (among other things) rule functions that will analyze collected data in order to figure out what wrong. The interface to the rules will be the same regardless of architecture.

[00125] The Monitor will have its own separate shell. The primary purpose of this shell is to display the Monitor's analysis to the user, and to allow the user to view certain state information. Thus, the Monitor's shell will be very simple, without many of the features found in the kernel's shell. It is expected that a novice user would simply look at the Monitor's FRU or FRU list, and act accordingly, without any Monitor shell interaction. But, expert users may wish to view more detailed information about the failure state, and perhaps the Monitors decision process.

[00126] KERNEL

[00127] The kernel will have to be modified primarily to allow the applications to do a better job at diagnosing. The goal, however, is to minimize the modifications as much as possible. The following list shows the areas of the kernel that will need to be modified:

[00128] Initialization (for the Monitor)

[00129] Initialization (for Drivers)

[00130] Monitor interface changes (see kernel interfaces section below)

[00131] New diagnostic capability

[00132] Processor Affinity (if it does not already exist)

[00133] Physical Memory Allocation

[00134]

[00135] MAJOR INTERFACES

[00136] There are 2 groups of interfaces that are to be considered in this document:

[00137] External Monitor interfaces (Monitor/Kernel and Monitor/Application)

[00138] Internal Monitor interfaces

[00139] Kernel/Application interfaces

[00140] EXTERNAL MONITOR INTERFACES

[00141] There will be four basic ways in which the Monitor, (RAGEMON) can be accessed from the kernel:

[00142] 1) Traps/Interrupts

[00143] 2) Kernel Panics

[00144] 3) Explicit Monitor calls

[00145] 4) Explicit Monitor error calls.

[00146] The following sections give more detail on each one of the access methods. Note, that for all the methods, the Monitor will be in real mode, with most traps, disabled.

[00147] TRAPS/INTERRUPTS

[00148] At various times, while the kernel is running, it may encounter a trap, fault, interrupt, or check. Not all of these conditions indicate an error condition.

However, any such condition will cause the kernel to enter the Monitor. This is because the kernel's trap handler will point to the Monitor's handler. This will ensure that the Monitor will capture any of these significant trapping conditions. In the case of multiple processors, the Monitor will either, have to provide a separate handler for each processor, or use semaphores to keep the processors from overwriting any critical pieces of Monitor code. In both cases, the Monitor will have to know how many processors there are.

[00149] During initialization, the kernel will have to inform the Monitor of the addresses of the routines, which handle the non-error cases. When one of these non-error traps occurs the Monitor will then branch to the appropriate routine, in such a way as to not alter the register state for the kernel.

[00150] KERNEL PANICS

[00151] Kernel panics generally occur when the kernel encounters some illegal situation, or when it cannot resolve some problem (e.g. page fault that can't be resolved). Kernel panics could result from software errors or hardware errors. The Monitor needs to know when they occur, so that it can make the distinction. All "Panic" calls in the kernel will be implemented using a variant of the break instruction (e.g. break 2). This will force the kernel into the Monitor, where the Monitor can determine the appropriate action for the panic.

[00152] MONITOR CALLS

[00153] During initialization, the kernel will need to pass information to, and perhaps get information from the Monitor. This type of kernel/Monitor communication is not related to errors. For these types of calls, the Monitor will not need to determine error conditions or try to freeze the kernel. Since the Monitor will be entered for any trapping condition, we can use another break instruction variant for a standard monitor call (e.g. break 3).

[00154] MONITOR ERROR CALLS

[00155] There will be cases where an application detects a hardware error condition directly. The application can use a system call to inform the kernel. The

kernel, in turn, can let the Monitor know via a Monitor error call. Unlike a standard Monitor call, the error call would indicate that the Monitor should freeze the kernel, collect error information, and begin fault analysis. Again, the use of a break variant would be convenient (e.g. break 3) in keeping a consistent interface to the Monitor. It may be better to allow applications to directly invoke these calls, so that the Monitor can gather "fresh" information about the error. Break instructions allow the Monitor to have the closest "snap-shot" of the register state, of an error event. Note that the break instruction takes an immediate field. This field will be use to distinguish between different types of Monitor calls.

[00156] MONITOR FREEZE CALLS

[00157] When the Monitor has been informed of an error condition, it will need to "freeze" the kernel. "Freezing" essentially means stopping all kernel processes. This feature is useful to fault analysis, for the case where one process interfered with another. The process that was interfered with could cause a trap. In which case, it would be useful to have information about the process that interfered with the trapping one. The more information one could get about the interfering process, at the time the trap occurred, the more useful that information would be.

[00158] On a single processor system, the kernel is essentially "frozen" when a process traps to the Monitor. This is because no other processes can run, since there's only one CPU, hence one thread. But, for multiple processor systems (MP systems), the other processors will have to be stopped. To accomplish this, the Monitor could issue a "Transfer Of Control" (TOC or reset command), to force all processors to a known place. The TOC would effectively "freeze" the kernel.

[00159] The Freeze call would be the only call, describe so far, where the Monitor is initiating communication to the kernel, rather than the other way around. For MP systems, it would be useful if the Monitor could switch to a non-trapping processor, in case the trapping one was severely mal-functional.

[00160] By default, if an MCA, or CMC is encountered, firmware will take over before returning to the Monitor. Firmware will then force all processors to collect state, and "rendezvous", before returning to the Monitor on the PD Monarch

processor. For other errors, where no firmware intervention is involved, the Monitor will have to issue a special external interrupt itself (via library routines) to "freeze" the other processors.

[00161] INTERNAL MONITOR INTERFACES

[00162] Unlike the Monitor/kernel interface, communication between the Library and the Monitor will occur through procedure calls. And, since the Library and Monitor are separate executables, there has to be some sort of stubbing mechanism between the two, in order for communication to occur. Recall that the Library is a "box" specific module that can change when RAGE is run on different boxes. Therefore, there will need to be some sort of standard interface between the Monitor and the library, so that the Monitor will not have to know which library is present. There will be a standard set of calls that the Monitor can make to the library, which will allow the Monitor to get the information it needs regardless of which "box" the library is written for.

[00163] The library has several functions that it has to perform for the Monitor. These functions can be broken down into several areas: Initialization, Error recovery, Error collection, and Fault analysis. On start-up, the loader (Rageldr) will load the Monitor first. Then, the appropriate library module will be loaded. The loader will pass the library module the address of the Monitor's caller entry point. The loader will pass control to the library module so it can initialize with the Monitor. Once that initialization is completed, control will pass back to the loader so that it can load the kernel.

[00164] The Monitor and the Library will communicate via a "stub" interface. There will be a standard set of index numbers that represent calls. The library will have stubs that specify the index of the desired call. The Monitor will have similar stubs so that the library and Monitor can communicate with each other, with either as the initiator.

[00165] Note, all Monitor functions will start with the prefix "Mon_", while all library functions will start with the prefix "Lib_".

[00166] INITIATILIZATION FUNCTIONS

[00167] When the loader first loads the library, and temporarily passes control to it, the library will need to pass its caller entry point address to the Monitor. The loader will have passed the Monitor's caller entry point to the library. Thus, the library can call the Mon_Init()function to pass its caller entry point to the Monitor. This call will also informed the Monitor weather to use the default "firmware intervention" method for severe errors, or the "back door" method.

[00168] The Lib_Chksum function does an additive checksum over the entire body of the library. This function is design to ensure that the library has been loaded into memory properly. The checksum result should be zero. Thus, the return of this function should be zero.

[00169] The Mon_Chksum()function is similar to the Lib_Chksum() function, except it allows the Monitor to ensure that it was loaded into memory properly.

[00170] The Mon_ProcCnt()function can be called by the library once kernel initialization is completed. This function will return the total number of processors detected by the kernel. If this function is called before the kernel initializes with the Monitor, or if the kernel fails during initialization, this function will return 0.

[00171] ERROR RECOVERY FUNCTIONS

[00172] The Mon_Grs(proc_id)function will return the general register state of the specified processor, at the time it last entered the Monitor, or at the time of the error (if the Monitor sensed an error condition). This function takes the processor ID as an argument. Whenever a processor enters the Monitor, its register state will be saved. When a processor re-enters the Monitor, the saved register state will be overwritten. The library routines can use this function to get the last known state of a processor. For error conditions, this routine will get the register state from firmware, if the default firmware intervention method is being used. For "back door" libraries, the Monitor will rely on a special LIB_GRs(proc_id) routine to get this information.

[00173] The Mon_Crs(proc_id) function is similar to Mon_Grs , but it returns the architectured control register state.

[00174] The Mon_Shutdown() function allow the Monitor to do any necessary clean up before the kernel stops running, and returns to the loader. The Monitor will clear any run flags in NVRAM when this function is invoked.

[00175] The Lib_Spec(index) function allows the Monitor to call non-standard library functions.

[00176] The Lib_Rcv(trap_id) function is called by the Monitor whenever a trap has occurred, and the register state has been saved. This library function will "recover" the trap so that the processor can still run. For HPMCs and LPMCs, if the default "firmware intervention" method is used, this function will do nothing.

[00177] Otherwise, this routine will save away any hardware specific error registers that it plans to alter. For most traps, this routine will effectively be a "no-op", but for traps deemed severe by the library, this routine will do the necessary saving and "clean-up".

[00178] The Lib_Freeze()function is called by the Monitor when the Monitor wants to stop all kernel processes and record the kernel state. Since doing this could vary from box to box, this routine allows details to be abstracted from the Monitor. By default, for nay error other than HPMCs and LPMCs, this function will invoke a TOC.

[00179] The Lib_SwitchProc(proc_id) function allows the Monitor to start executing on the specified processor. Certain error conditions might make the processor, that the Monitor is currently running on, unstable. In such cases, if the system is a multi-processor system, it would be advantageous to have the Monitor switch to a more stable processor.

[00180] The Lib_ProcID()function allows the Monitor to obtain the processor ID of the currently running processor.

[00181] ERROR COLLECTION FUNCTIONS

[00182] The Lib_Collct()function is called by the Monitor once it determines that an error condition has occurred on the currently running processor. This routine will collect all register and other information that the Monitor does not collect by

default. This function masks the Monitor from having to know about any "box" specific registers whose state needs collecting. By default, for HPMCs and LPMCs, this function will make firmware calls to obtain certain error information.

[00183] FAULT ANALYSIS

[00184] The LIB_FaultAnal() function is the central function involved in fault analysis. The Monitor will call this function once all the information is collected for all processes. This function points to the rules list that the Monitor will process to determine the failing FRU.

[00185] The Mon_Printf(...) function allows the Monitor to print to the console. It has the same parameter signature as a standard printf call.

[00186] The Mon_KernelSym(address) function will return the name of the kernel function that contains the specified address.

[00187] KERNEL INTERFACES

[00188] There are several changes to the standard kernel that will have to be made, in order to allow the Monitor to do its job:

[00189] 1) Monitor Initialization

[00190] 2) Heartbeat Functionality

[00191] 3) Timed Semaphores

[00192] 4) Explicit Monitor calls

[00193] 5) Driver Modification

[00194] These kernel modifications will need to be made in such a way so that if the Monitor is not present, no Monitor contact will be attempted. When the loader launches the kernel, it will pass the Monitor's trap handler address, if the Monitor is present. Otherwise, a zero will be passed. In this way, the Kernel will know whether the Monitor is available.

[00195] MONITOR INITIALIZATION

[00196] When the Rage kernel begins to boot, if it detects the Monitor's presence, it will need to initialize with the Monitor. Initialization involves specifying to the Monitor certain kernel specifics, such as the following:

[00197] 1) The kernel symbol table

[00198] When the loader loads the kernel, it will also load the kernel symbol table. A pointer to this table will be given to the Monitor so that the Monitor will have access to all of the kernel's global addresses. This allows the Monitor to analyze the kernel, even when it fails to come up.

[00199] 2) The kernel routines for specific types of traps

[00200] The kernel will specify to the Monitor all the traps it wants to handle itself. The kernel will have to pass the address of the routines its whishes called upon encountering certain traps. The Monitor will save the addresses, so that when a trap occurs, the Monitor will call the appropriate, pre-assigned routine.

[00201] 3) The kernel start of free memory

[00202] If the Monitor knows the kernel's start of free memory, it can determine whether certain errors are related to compile time data structures or data structures that were created at run-time. In addition to the start of free memory, it would be useful for the Monitor to know the location of certain run-time structures (e.g. the map structures of the page allocator).

[00203] 4) The number of processors

[00204] The Monitor obviously needs to know the number of processors. This is important, for instance, when the Monitor is freezing. It will need to know when all of the processors have checked in. Also, there are cases where the Monitor may want to switch processors to avoid hardware problems on the first trapping processors. The Monitor would like to know the state of all processes that were running at the time the failure occurred. Since every processor can be running a process, all running

processors must be known. The Monitor could determine the number of processors itself, but since this has to be done by the kernel, why duplicate the code in the Monitor? Therefore, the Monitor will rely on the kernel for this information.

[00205] HEARTBEAT FUNCTIONALITY

[00206] Certain error conditions could cause the kernel to get stuck in some kind of infinite loop. On normal operating systems, these kinds of errors are difficult to diagnose. Ideally, it would be useful for the kernel to periodically "check-in" with the Monitor so that the Monitor can detect a hanging condition. The "Heartbeat" functionality of the Monitor is designed for precisely this situation. The implementation of the heartbeat functionality is more difficult on a uni-processor system than a multi-processor one.

[00207] Due to the periodic nature of the heartbeat function, the interval timer interrupt is the likely candidate for the "heartbeat" function. Every time a processor has a timer interrupt, the Monitor will be called. The Monitor can then check how many interval timer ticks have passed since the last interrupt. If more than, say three, time slices have passed, the Monitor would consider this a potential error condition (Note, that the exact number of time slices to be considered as an error will be determined heuristically, based on the kernel behavior). The Monitor would look at the address queues (as directed by the rules from the library) to see if the kernel was executing in the same general vicinity as it was during the last timer interrupt. If so, this would cause the Monitor to freeze the system and indicate a kernel hang. The Monitor would then try to further analyze the hang (i.e. find out what routine was hanging, etc.).

[00208] On a multi-processor system, it is unlikely that all the processors will hang at the same time. Therefore, some processor will enter the Monitor, sooner or later. At that point, the Monitor can check on the processor that had the timer interrupt, as well as the other processors. If the interval timers of all processors have been synchronized (this can be done via diagnose instructions) then, the Monitor can determine whether some other processor has taken too long to "check-in" with the Monitor. Due to the passive nature of the Monitor, a uni-processor system could hang

and be undetected by the Monitor. If the infinite loop that the kernel is in, has the interval timer interrupt turned off, and no other types of traps occur within the loop, the Monitor will not be called!

[00209] TIMED SEMAPHORES

[00210] Semaphore functions are prime examples of code where the kernel could hang, without any interrupts. Such a hang would be especially dangerous on a uni-processor system since the Monitor would never be called! In order to reduce this possibility, all semaphore functions in the kernel should be timed. In other words, the all semaphore functions should only try to access a semaphore for a fixed amount of time. Once that time elapses, the semaphore function should call the Monitor.

[00211] EXPLICIT MONITOR CALLS

[00212] The kernel can call the Monitor directly using an explicit Monitor call. As previously mentioned, the kernel will call the Monitor during initialization. However, there are many situations where the kernel may want the Monitor to track some information. For example, it would be useful for the Monitor to know which processor had a particular semaphore at any given time. That way, if some other processor timed out on that semaphore, the Monitor would know who had it. Also, it would be useful for the Monitor to know when a particular application is being run by a processor. If that processor trapped out to the Monitor, the Monitor could then know to use the applications symbol table, rather than the kernels, for analysis.

[00213] Explicit Monitor calls lie at the heart of what is meant by "instrumenting" the kernel. These explicit calls allow the kernel to pass information to the Monitor that the Monitor can use in fault analysis. This is a powerful feature. Most ordinary kernels would not be instrumented in this way because of the hit to performance that would be encountered. The RAGE kernel, however, trades off the performance hit, for the ability to track the kernels activities for debugging. The rules in the library will use this tracking feature along with other features, to do fault analysis. The other features would include things like: Using the symbol tables in the ELF format to find the names of routines; Using a feature of the calling convention where calls to sub-routines are always made using gr2 has the return address.

[00214]

[00215] DRIVER MODIFICATION

[00216] In order to facilitate faster kernel boot, many of the kernel drivers may have to be modified. On large systems with many I/O cards, kernel boots could be long due to the initialization required for all the I/O devices. Traditionally, the kernel will ensure all I/O devices are initialized before the user can interact with the console. For RAGE however, a delayed I/O initialization could suffice. All drivers, except the core essentials, could be made into LINUX modules. So that, their initialization could be delayed, allowing the user to have console access and the ability to launch RAGE applications that don't require the un-initialized I/O. Once the modules are initialized, RAGE apps can be launched to stress the I/O associated with them. The details of RAGE I/O are a part of the second RAGE development phase. Once the RAGE I/O investigation has been completed, the associated investigation report will give more details on RAGE I/O.

[00217] PERFORMANCE CONSIDERATION

[00218] Though the Monitor intervention will slowdown the Kernel performance to some degree, it should not so severely degrade the Kernel as to reduce the stress impact of the RAGE applications.

[00219] MODULE DETAILS

[00220] Once an error condition has occurred and the error information has been collected, the Monitor will need to begin Fault analysis. To do this, the Monitor will use a rules-based expert system methodology. The rules will be located in the library module. Conceptually, each rule will look something like:

[00221] IF (condition0) (condition1) .....(conditionN) THEN (action0) (action1).....(actionN)

[00222] There will be a set of such rules in the library module. The rules will be ordered in such a way as to speed up rules processing. In other words, rules that are more likely to be triggered early, will be placed near the start of the rules list. To start

Fault analysis, each rule in the rules list will be evaluated sequentially. The conditions in the "IF" part of a rule will actually be function pointers. Each function pointed to will either return 0, or a non-zero value. Zero will represent a "FALSE" condition. When a rule is being evaluated, if the condition is non-zero, it will be a function pointer. That function will then be executed, and the result checked. If the return value of the function is zero, no further conditions of the rule will be evaluated. The rule is then said to be NOT-TRIGGERED. Furthermore, if any of the conditions are zero (i.e. the function pointer is "NULL"), that condition is considered to be FASLE. And, therefore, the rule would also be NOT_TRIGGERED.

[00223] If all the conditions of a rule return non-zero, that rule is said to be TRIGGERED. The Monitor will keep evaluating each rule in the rules list, until it gets to a rule that is TRIGGERED. If none of the rules are triggered, the Monitor will indicate this condition and halt. Otherwise, the Monitor will attempt to execute each of the actions specified by the TRIGGERED rule. Once a rule has been TRIGGERED, the Monitor will not evaluate any further rules in the rules list. Also, any trigger rule will be disabled so that it cannot be TRIGGERD again. This feature is intended to speed up rules processing.

[00224] The actions of a rule are also function pointers. The Monitor will call the function of each action in a TRIGGERED rule. Each action function will be called sequentially, starting with the first. Once all of the action functions have been called, the Monitor will disable the rule and then start from the beginning of the rules list, looking for the next TRIGGERED rule. See the Data Structure section above for an example of what a rule would look like.

[00225] Each rule should have a short text string that basically describes what the rule is trying to due. This feature allows the Monitor to display, in English, to the novice user how it came to its conclusion, during analysis.

[00226] There are several debugging features that the Monitor can take advantage of in order to determine the failing FRU:

[00227] Kernel Symbol Table

[00228] The Kernel will be an ELF-64 binary. As such, it has its symbol table embedded in the binary. The library rules can use this feature to determine where, in the kernel, an error occurred. Applications will also be compiled into ELF-64 formats. Thus, the same technique can be used. For instance, if the address, at which a particular processor failed, is known, the function that it was executing at the time can be determined from the symbol table. Since the symbol table contains all the starting address of the functions contained in the binary.

[00229] Procedure stack unwind

[00230] The Monitor can take advantage of un-winding the stack to trace procedures backwards thru time.

[00231] IMPLEMENTATION WITH A PRODUCTION OS

[00232] In the discussion above, the inventive diagnostic monitor is configured to operate with a non-production OS. As discussed, a production OS is typically optimized for performance and/or security and perhaps compatibility, not diagnostic. Accordingly, diagnostic exercisers operating under a production OS are typically severely restricted in their ability to gather data, to detect errors, and to perform error analysis.

[00233] With the inventive diagnostic monitor, it is contemplated by the inventor herein that there may be situations wherein it is highly desirable to have an independent diagnostic binary operating cooperatively with the production kernel of a production OS. Unlike the prior art diagnostic routines which tend to run as privileged processes or drivers under the production kernel, the inventive diagnostic monitor is preferably configured such that it is loaded even before the kernel is loaded. In this manner, the inventive diagnostic monitor can provide error analysis data even if the kernel could not be loaded.

[00234] Furthermore, even though the inventive diagnostic exerciser operates cooperatively with the production kernel, the inventive diagnostic monitor executes independently of the kernel and thus can perform error analysis in the event of a kernel crash. Thus, the time-consuming process of dumping the kernel and the labor-

intensive process of sorting through the kernel dump data to analyze the error that causes the kernel to crash are advantageously avoided.

[00235] In accordance with one embodiment of the present invention, the production OS is minimally modified to accommodate the inventive diagnostic monitor. In particular, it is desirable to minimize the effort and/or degree of modification to the production OS, both to minimize the amount of work involved in the modification and also to minimize affecting the performance and/or security features of the production OS. While the monitor is clearly capable of handling more routine errors, the performance tradeoff needs to be taken into consideration. Thus, it is contemplated by the inventors herein that the monitor is involved, in various embodiments herein, only in situations that involve severe kernel errors, i.e., kernel errors that can cause the kernel to hang or crash. These severe kernel errors include, for example, severe hardware errors, kernel panics, and timed-out semaphores.

[00236] Fig. 8 depicts, in accordance with one embodiment of the present invention, a diagnostic arrangement that employs the inventive diagnostic monitor in conjunction with a production OS. The arrangement 804 includes a loader module 805, which is responsible for loading major components of diagnostic monitor 808 into memory upon system initiation. These major components are discussed hereinbelow.

[00237] Arrangement 804 includes a production kernel 806, representing the kernel of the production OS (e.g., a production Windows™, Unix, or Linux OS) employed in the field. There are provided with production kernel 806 a plurality of drivers 807a, 807b, 807c, representing some of the I/O drivers provided with kernel 806.

[00238] Diagnostic monitor 808 is designed to communicate with and to monitor production kernel 806. In one embodiment, monitor 808 and kernel 806 are furnished as separate binary files, thereby rendering monitor 808 independent of the OS (i.e., monitor 808 runs with rather than runs under kernel 806). Monitor 808 is preferably loaded into a memory location that is unused by kernel 806 to avoid any unintended alteration of monitor 808 by kernel 806. When a severe error occurs,

monitor 808 will attempt to analyze the error and isolate the problem to a FRU (field replaceable unit) or a list of potential offending FRUs. In this sense, the inventive arrangement provides the error isolation granularity typically associated with off-line diagnostic tools.

[00239] As shown in Fig. 8, diagnostic monitor 808 includes three main modules: monitor logic 810, monitor library 812, and monitor trap table 814. Monitor logic 810 is architecture-dependent (e.g., PARISC™, Itanium™, etc.). Depending on the specific architecture and the type of box of computer hardware 802 (the information pertaining to which may be obtained via link 816), the appropriate monitor library 812 would be loaded. Monitor library 812 is thus both architecture-specific and box-specific in one embodiment.

[00240] Monitor trap table 814 represents the trap table that is in control during execution of the diagnostic applications. As will be discussed later herein, one of the major functions of diagnostic monitor 808 is to control the trap handler. Any trap encountered by kernel 806 would be examined by monitor 808, which will decide whether it will handle the trap itself or will pass the trap on to the kernel for handling. Preferably, this examination process is optimized such that only severe errors (e.g., kernel panics, severe hardware error, or timed-out semaphores) are handled by the diagnostic monitors and other traps and/or errors are quickly ascertained and handled by the kernel's own trap table..

[00241] The kernel has a trap table, and the kernel's trap table is granted control if diagnostic monitor 808 decides that it wants the kernel to handle a particular trap. In such case, control will pass to the appropriate offset in the trap table of the kernel to handle the trap. Thus, as far as the kernel is concerned, its own trap table gets the traps that the kernel needs to handle in a transparent manner. Other traps, e.g., those related to severe errors, are retained by and handled by monitor trap table 814.

[00242] Applications 830, 832, 834, and 836 represent some user applications that may be executed under kernel 806 during the course of normal execution. In contrast with the diagnostic applications discussed earlier, these user applications

represent applications executed by users in the normal course of employing the computer system. These user applications may include, for example, database applications, spreadsheet programs, graphics programs, web-related programs, and the like.

[00243] Fig. 9 illustrates the diagnostic monitor 808 in greater detail in accordance with one embodiment of the present invention. As can be seen, monitor logic 810 includes, in addition to core monitor codes 914, architecture routines 910 and rules engines 912. Although architecture routines 910 and rules engines 912 are shown separate from monitor logic 810 in Fig. 9 to facilitate discussion, they are in fact part of monitor logic 810. Architecture routines 910 represent routines that may perform standard tasks based on a given architectural level. For instance, the contents of all general registers may need to be saved. This would be done by an architectural routine. The size and number of general registers depends on the particular architecture under consideration (e.g. PARISC 1.1 has 32 bit width registers, while PARISC 2.0 has 64 bit width registers). In one embodiment, although the core monitor is dependent on the architecture, different monitor libraries are provided to accommodate different boxes, thereby enabling the core monitor to be employed for all boxes pertaining to a given architecture.

[00244] Monitor trap table 814 is designed to be the functional trap handler during execution. Thus, any trap/interrupt by kernel 806 (see Fig. 8), expected or not, will force it to enter diagnostic monitor 808. While routine traps and errors are returned to the kernel's trap table for handling, severe traps/errors are handled by the diagnostic monitor itself. For example, for severe hardware errors, significant software errors, and hardware resets, the hardware may automatically branch to firmware routines. In the context of PARISC™ machines, for example, HPMCs (High Priority Machine Checks), LPMCs (Low Priority Machine Checks), and TOCs (Transfer of Controls) may cause the hardware to automatically branch to firmware routines. The firmware may then save the register state of the processor and clear any error conditions. If the appropriate trap handler routine in diagnostic monitor 808 is setup properly, the firmware routine may branch to diagnostic monitor 808's handler. At this point, diagnostic monitor 808 can then begin analysis.

[00245] Rules engine 912 of monitor logic 810 includes a set of routines to interrogate monitor library 812 in order to determine how to find the FRU for the error condition. Rules engine 912 is employed to interpret rules 920 (described below), and may also include functionality such as which rules will be employed for error analysis and when error analysis is completed. There may be a library for each type of "box" for a given architecture. The library may contain (among other things) rule functions that may analyze collected data in order to figure out what went wrong. The interface to the rules may be the same across different architectures.

[00246] Monitor library 812 includes, in addition to core library codes 924, rules 920 and hardware-specific routines 922. Rules 920 represent rules specific to the box and architecture under consideration and are employed for error analysis. Rules 920 has the inferences for various error conditions and can resolve the error to the failing FRU. Hardware-specific routines 922 include routines specific to a particular box to facilitate, for example, the gathering of any necessary information from the specific box.

[00247] Upon initialization of the computer system, loader module 805 loads the components of diagnostic monitor 808 and production kernel 806 in a particular sequence. The sequence is analogous to that described in Figs 3A and 3B and will not be redundantly repeated here. As will be apparent to those skilled in the art, it is necessary to modify the production kernel so that the diagnostic monitor, if present, can be loaded first. Furthermore, it is preferable that the production kernel does not automatically take control of the trap table upon system initialization. In one embodiment, the modification involves storing the base address of the diagnostic monitor's trap table (instead of the base address of the kernel's own trap table) in the register that is designated for storing such data.

[00248] Figs. 10A and 10B illustrate, in accordance with one embodiment of the present invention, the error handling steps by the diagnostic monitor. Because a trap could be a result of normal operation or it could represent a problem, the diagnostic monitor needs to examine traps and make certain decisions regarding how the traps are handled. In block 1002, the diagnostic monitor ascertains whether the trap is to be handled by the kernel or by the diagnostic monitor itself. In one

embodiment, the decision is made using a bit masking technique for looking at the bit in a particular position of the variable. However, optimizations may be made to minimize impact on the performance of the production OS. By way of example, the traps to be handled by the monitor can be enumerated so that the code can branch to the appropriate address and offset in the monitor's trap table automatically. Alternatively or additionally, the traps to be handled by the kernel's own trap table can be enumerated so that the code can branch to the appropriate address and offset in the kernel's trap table automatically.

[00249] If the trap is a result of normal operation, such as a page miss, such trap may be handled by the OS's kernel. In that case, the method proceeds to block 1004 wherein it branches to the appropriate OS trap table offset using the symbol table loaded earlier (e.g., in block 340 of Fig. 3) and/or the information obtained during diagnostic initialization call by the kernel (e.g., in block 314) and/or via hard-coded branching instructions.

[00250] For example, in the PARISC™ implementation, branching can be performed by, for example, using a relative branch instruction (e.g., branch to five instructions away) or by giving the actual address to branch to in the branch instruction. Alternatively, branching may be accomplished by reference to a general register, which contains the address.

[00251] However, for Itanium-based systems, if a branch instruction employs a branch register, there is a potential for inadvertently overwriting the content of a branch register that is in use by the kernel since the kernel also employs branch registers. It is possible to force the kernel to temporarily store and restore the values of these registers but that may involve certain additional modifications to the production kernel.

[00252] There is provided, in accordance with one embodiment of the present invention, a branch table within the diagnostic monitor. With reference to the IPF™ implementation, for example, the branch table is initially setup so that all branches jump to themselves. During initialization, the monitor can dynamically calculate the address of the long branch instructions and can alter all the branches of the branch

table to branch to the appropriate offset in the kernel trap table. This is possible because the kernel trap table resides at a fixed address.

[00253] To clarify, a long branch is a type of relative branch that allows one to branch to a long offset, typically longer than normally performed using a normal relative branch. Using long branches allows the monitor to be positioned at any desired location in memory, and still be able to get to the kernel trap handler. Otherwise, the monitor would have to reside near the kernels trap handler in memory. Furthermore, the monitor long branch table renders it possible to avoid using the branch registers, which are utilized by the kernel. Thus, it is not necessary to keep track of the contents of the branch registers, which would require more kernel modification. For implementations that do not involve branch registers, for example PARISC™, general registers may be employed instead.

[00254] Returning to Fig. 10A, if the trap is to be handled by the diagnostic monitor (as ascertained in block 1002), a decision is made in block 1006 whereby the diagnostic monitor ascertains whether the trap is an explicit monitor call by the kernel. By way of example, the immediate values associated with break trap instructions, which are employed to monitor calls, may be specified to allow the diagnostic monitor to ascertain whether a trap is an explicit monitor call by the kernel.

[00255] If the trap is an explicit monitor call, it is handled in block 1040. In this case, an explicit monitor call has been received. The application is requesting that the monitor perform some function (e.g., set the error logging threshold or return correctable error information). The monitor will then perform the function and return to the application.

[00256] As mentioned, only severe errors are preferably handled by the diagnostic monitor executing in cooperation with a production kernel. A HPMC (PARISC™) or a MCA (IPF™) represents an example of a severe error that causes explicit types of traps. An example of a severe, non-trapping type of error may be, for example, a semaphore timeout. In a semaphore timeout, some processor has grabbed the semaphore for longer than some certain threshold, which situation indicates an imminent hang. In this case, the semaphore may timeout, for example, and result in a

call to the monitor. Generally speaking, the diagnostic monitor would treat the severe error monitor call in the same manner that it treats kernel panics. The steps to handle kernel panics is discussed in connection with Fig. 11 herein.

[00257] If the trap is not an explicit monitor call (as determined in block 1006), it is deemed a severe error. As an example, the failure of kernel boot is a severe error. As mentioned before, severe hardware errors are one class of sever errors to be handled by the diagnostic monitor. In IPF™, for example, the occurrence of a Machine Check Abort (MCA) condition is typically regarded as a severe error. In PARISC™, for example, the occurrence of a High Priority Machine Check (HPMC) condition is typically regarded as a severe error. Generally speaking, each architecture has its own way of representing a severe error. However, details on the error can vary between different box types. For example, there may be one bit in a given hardware register in the processor which will be set if the error is severe. Generally speaking, the error details may be ascertained using the rules.

[00258] With a severe error, the diagnostic monitor then stops the kernel from running by sending an interrupt signal to all other processors (assuming there are multiple processors). It is possible, for example, for one processor to experience a severe error (e.g., a HPMC in the case of PARISC™) but other processors of the computer system may not have experienced the same severe error. Freezing the processors by an external interrupt signal allows the state of the processors to be collected in block 1012 to facilitate analysis in block 1014. Of course if there is only one processor in the system, there is no need to send the interrupt signal to the non-existent other processors, in which case block 1010 is not necessary.

[00259] In block 1014, the error analysis may be accomplished using, for example, the hardware-specific routines (822 in Fig. 8) of the monitor library. The hardware-specific routines may examine various registers of the processors, the I/O registers, certain data structures in the kernel, and the like, to ascertain the error and/or the cause thereof and preferably to narrow the error down to one or more FRUs (Field Replaceable Unit). For example, analysis may pinpoint that a particular CPU is defective and needs to be replaced. The analysis result is displayed in block 1016. In one embodiment, the result may be displayed in a monitor shell that is separate from

the OS's shell. The monitor shell permits certain diagnostic-related inputs and outputs and can continue to run even if the OS crashes. For example, one can employ the monitor shell to specify certain kernel data structures and routines and causing those to be displayed for examination. This is represented in block 1018 of Fig. 10.

[00260] Note that the rules and rules engine of the monitor allow it to analyze the relevant (box specific, in some cases) registers and data structures of the kernel (if necessary) and to narrow the problem to, for example, the failing FRU. This is in contrast to the prior art situation employing a standard OS, wherein an experienced user or expert would have to manually analyze the data dumped by the kernel. Furthermore, the monitor has access to all system and kernel data, thereby ensuring that the monitor can obtain any required information in order to accurately perform the error detection and error analysis tasks. For example, in one embodiment, the map file of the kernel may contain the addresses of the global kernel variables and kernel function addresses, all of which may be accessible by the monitor. In contrast, the data dump by the kernel may not, in some cases, contain all the required data for error analysis.

[00261] There are times when kernel panics occur without traps. For example, the kernel may have encountered some illegal situation where it runs out of resources (e.g., memory) or experiences a problem that it cannot resolve (e.g., an unresolvable page fault). In this case, the kernel may have made a panic call, which is converted into a monitor call by the modified production kernel to allow the diagnostic monitor to become involved. The kernel panic-related monitor call however contains information identifying the monitor call as one that results from a kernel panic. Once the panic-related monitor call is received, it is handled in blocks 1102-1110 of Fig. 11. Blocks 1102-1110 perform error handling in an analogous fashion to that performed by blocks 1010-1018 of Fig. 10B.

[00262] As can be appreciated from the foregoing, the use of the inventive diagnostic monitor that operates as an independent binary in cooperation with the production kernel enables the inventive diagnostic monitor to perform error analysis even in the event of a kernel crash or in the event that the kernel fails to load.

[00263] The fact that the monitor is involved only in severe error cases, and not employed to handle non-severe errors that occur during normal execution (e.g., page fault, correctable single-bit memory errors) also reduces the negative impact on system performance. During normal execution, none of the three cases that are discussed (e.g., severe hardware errors, kernel panics, time-out semaphores) would be encountered, and thus the impact of the monitor on system performance is minimal.

[00264] As mentioned, the above exemplary implantation is only one specific example of diagnostic monitor implementation in accordance with principles of the present invention. Thus while this invention has been described in terms of several preferred embodiments, there are alterations, permutations, and equivalents which fall within the scope of this invention. It should also be noted that there are many alternative ways of implementing the methods and apparatuses of the present invention. It is therefore intended that the following appended claims be interpreted as including all such alterations, permutations, and equivalents as fall within the true spirit and scope of the present invention.